In this representation, the nodes are stored level by level, starting from the zero level where only root node is present. Root node is stored in the first memory location (as the first element in the array).

Following rules can be used to decide the location of any node of a tree in the array (assuming that array index starts from 1):

1. The root node is at location 1.
2. For any node with index i, $1 < i \leq n$, (for some n):
   a) PARENT (i) = $\lfloor i/2 \rfloor$

      For the node when i = 1, there is no parent.

   b) LCHILD (i) = 2 * i

      if 2 * i > n, then i has no left child

   c) RCHILD (i) = 2 * i + 1

      if 2 * i + 1 > n, then i has no right child

ex: (A - B) + C * (D / E)

→ Order of evaluation, (A - B) + C * (D / E)

1 2 3 4

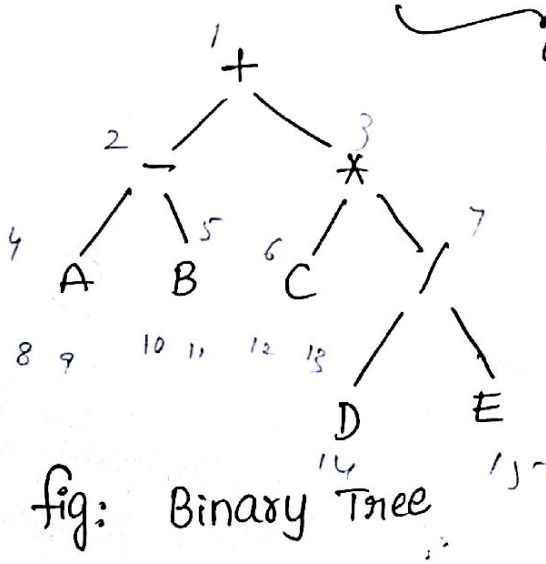



fig: Binary Tree



h = 4

array size = $2^h - 1$

$= 2^4 - 1 = 16 - 1 = 15$

→ Linear or sequential representation of the above example,

| a[i]: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | + | − | * | A | B | C | / | | | | | | | D | E |

consider a node with index i = 4, a[4] = A

parent(i) = ⌊i/2⌋ ✓

= ⌊4/2⌋ = 2

∴ parent(i) = a[2] = −

∴ parent of A = −

LCHILD(i) = 2*i

LCHILD(4) = 2*4 = 8 ✓

∴ left child(4) = a[8] = NULL

RCHILD(i) = 2*i+1

RCHILD(4) = 2*4+1 = 9

∴ Right child(4) = a[9] = NULL



How the size of an array can be estimated ?

A binary tree of height h can have at most $2^h - 1$ nodes. So the size of the array to fit such a binary tree is $2^h - 1$

Advantages :-

1) Any node can be accessed from any other node by calculating the index and this is efficient from execution point of view.

2) Here, only data are stored without any pointers to their successor or ancestor which are mentioned implicitly.

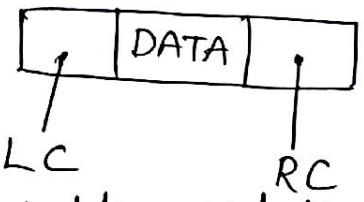
3) Programming languages, where dynamic memory allocation is

not possible (such as BASIC, FORTRAN), array representation is the only means to store a tree.

Disadvantages :-

1) Other than full binary trees, majority of the array entries may be empty.
2) It allows only static representation. It is no way possible to enhance the tree structure if the array size is limited.
3) Inserting a new node to the tree or deleting a node from the tree are inefficient with this representation, because these require considerable movement up and down the array which demand excessive amount of processing time.

2) Linked Representation :- Linked representation assumes the structure of a node as,




Here LC & RC are two link fields used to store the addresses of left child and right child of a node, DATA is the information content of the node. With this representation, if one knows the address of the root node then from it any other node can be accessed.

ex: (A-B) + C * (D / E)

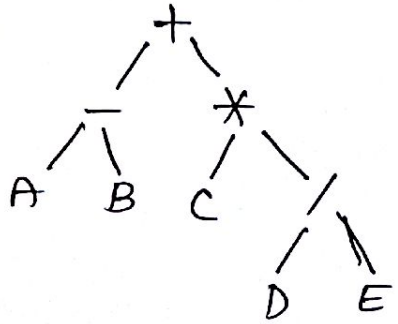



fig: Binary Tree

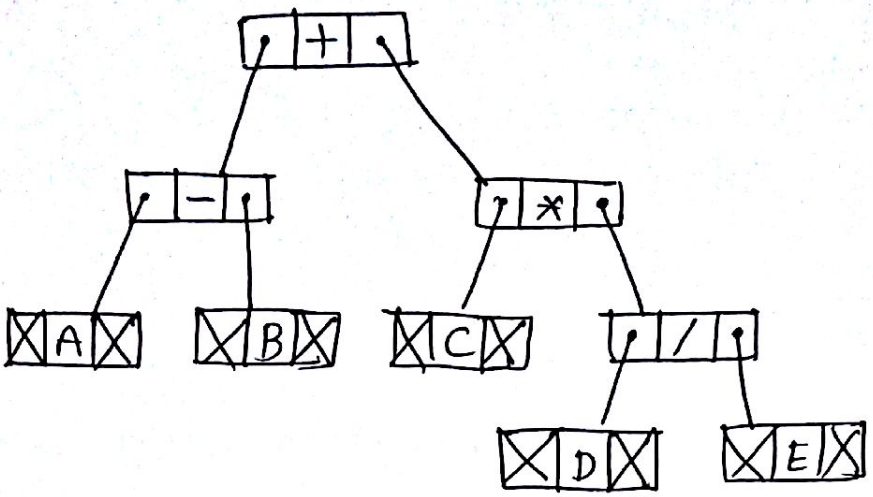



fig: Linked representation (logical view)

| Address | LCHILD | DATA | RCHILD |
|---|---|---|---|
| 50 | | A | |
| 75 | 50 | - | 40 |
| 40 | | B | |
| 89 | 75 | + | 46 |
| 62 | | C | |
| 46 | 62 | * | 58 |
| 66 | | D | |
| 58 | 66 | / | 74 |
| 74 | | E | |

fig: A binary tree & its various nodes (physical view)

Advantages: It allows dynamic memory allocation. Hence the size of the tree can be changed as and when the need arises without any limitation except the limitation of the availability of the total memory.

Disadvantages: The linked representation uses more memory than that required by the linear representation. Linked representation requires extra memory to maintain the pointers. Some pointers though with NULL values, they too need memory to store them. In a linked representation of a binary tree, if there are n no. of nodes then the no. of NULL links are $n+1$

→ Maximum & minimum size that an array may require to store a binary tree with n no. of nodes are

$$Size_{max} = 2^n - 1$$

$$Size_{min} = 2^{\log_2(n+1)} - 1$$

Operations on Binary Tree :-

1. Insertion : To include a node into an existing (may be empty) binary tree.
2. Deletion : To delete a node from a non-empty binary tree.
3. Traversal : To visit all the nodes in a binary tree.
4. Merge : To merge 2 binary trees into larger one.

Traversals (Recursive): Traversals operation is a frequently used operation on a binary tree. This operation is used to visit each node in the tree exactly once. A full traversal on a binary tree gives a linear ordering of the data in the tree.

Now a tree can be traversed in various ways. There are six such ways possible.

① $R\ T_l\ T_r$ ④ $T_r\ T_l\ R$

② $T_l\ R\ T_r$ ⑤ $T_r\ R\ T_l$

③ $T_l\ T_r\ R$ ⑥ $R\ T_r\ T_l$

Here $T_l$, $T_r$ denote the left & right subtrees of the node R.

visit 1 & visit 4, visit 2 & visit 5, visit 3 & visit 6 are mirror symmetric. So only there are 3 fundamental traversals as given below.

1. $R\ T_l\ T_r$ (preorder) DLR
2. $T_l\ R\ T_r$ (Inorder) LDR
3. $T_l\ T_r\ R$ (Postorder) LRD

L - left R - right D - Data

1. Preorder Traversal :- In this traversal, the root is visited first, then the left subtree in preorder fashion, and then the right subtree in preorder fashion. Such a traversal can be defined as follows:

1) visit the root node
2) Traverse the left subtree of R in preorder
3) Traverse the right subtree of R in preorder

## Algorithm Preorder:

Input : ROOT is the pointer to the root node of the binary tree

Output : Visiting all the nodes in preorder fashion.

Datastructure : Linked structure of binary tree.

Steps:
1. ptr = ROOT
2. if (ptr != NULL) then
3. Write ptr → DATA
4. Preorder(ptr → LC)
5. Preorder(ptr → RC)
6. end if
7. Stop

ex:

preorder traversal : D L R

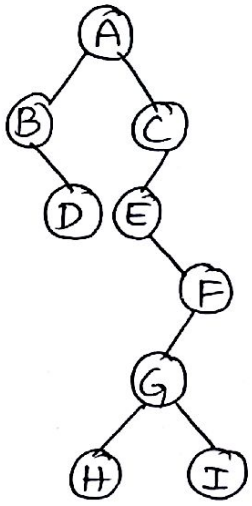



fig: Binary tree

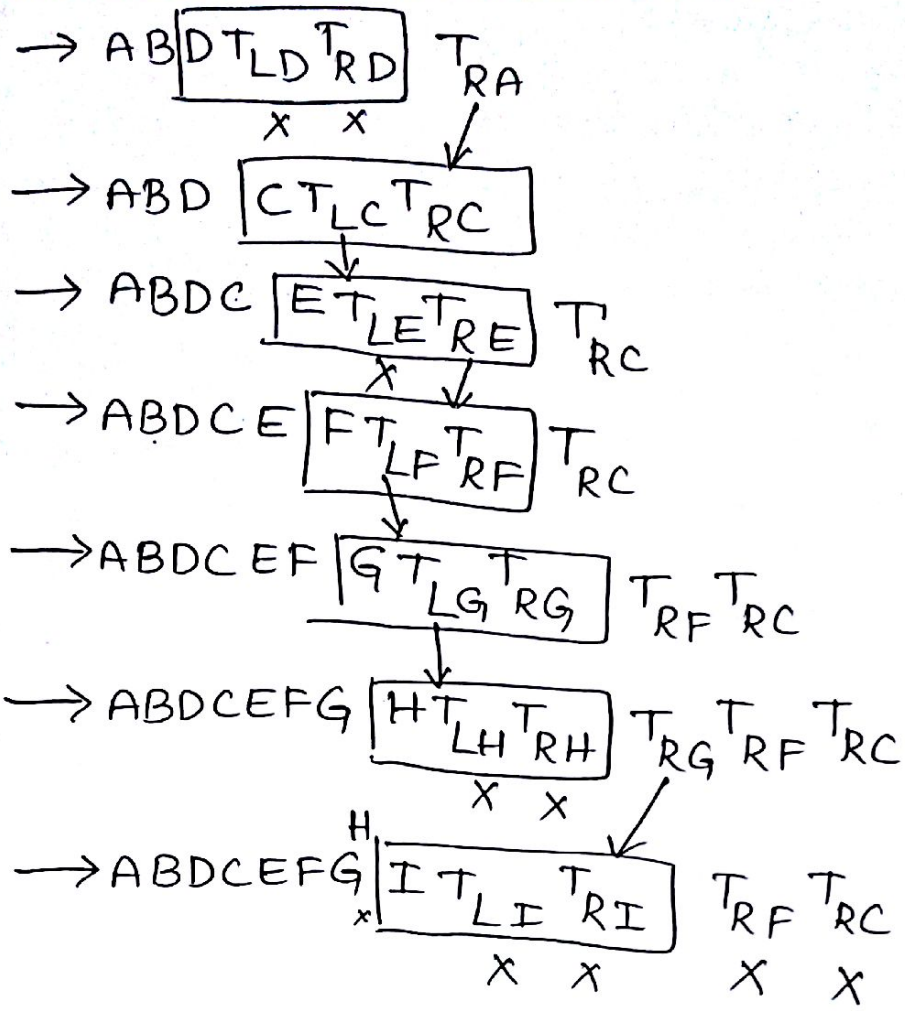
Traversing is done as follows:

→ $AT_{LA}T_{RA}$

↓

→ $A\boxed{B\,T_{LB}\,T_{RB}}\,T_{RA}$

X




∴ The preorder traversal for the given binary tree is ABDCEFGHI.

(2) Inorder Traversal:

→ Traverse the left subtree of the root node R in inorder.

→ Visit the root node R

→ Traverse the right subtree of the root node R in inorder.

## Algorithm Inorder:

Input: ROOT is the pointer to the root node of the binary tree.

Output: Visiting all the nodes in inorder fashion

Data structure: Linked structure of binary tree.

Steps:
1. ptr = ROOT
2. if (ptr ≠ NULL) then
3. inorder (ptr → LC)
4. Write ptr → DATA
5. inorder (ptr → RC)
6. end if
7. stop



ex:

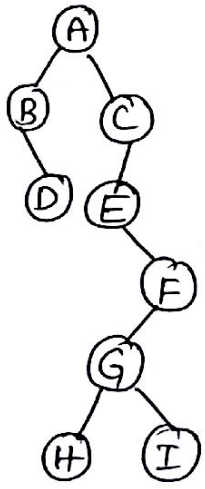
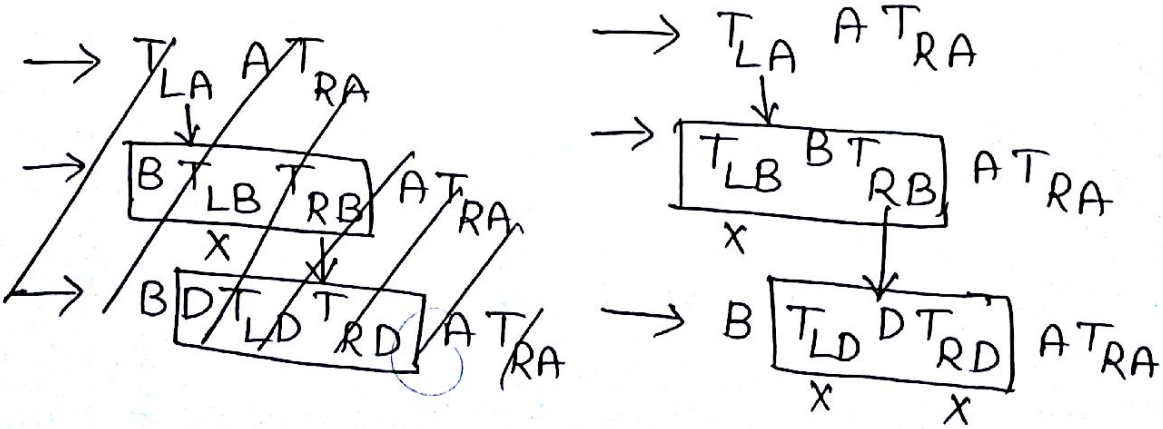
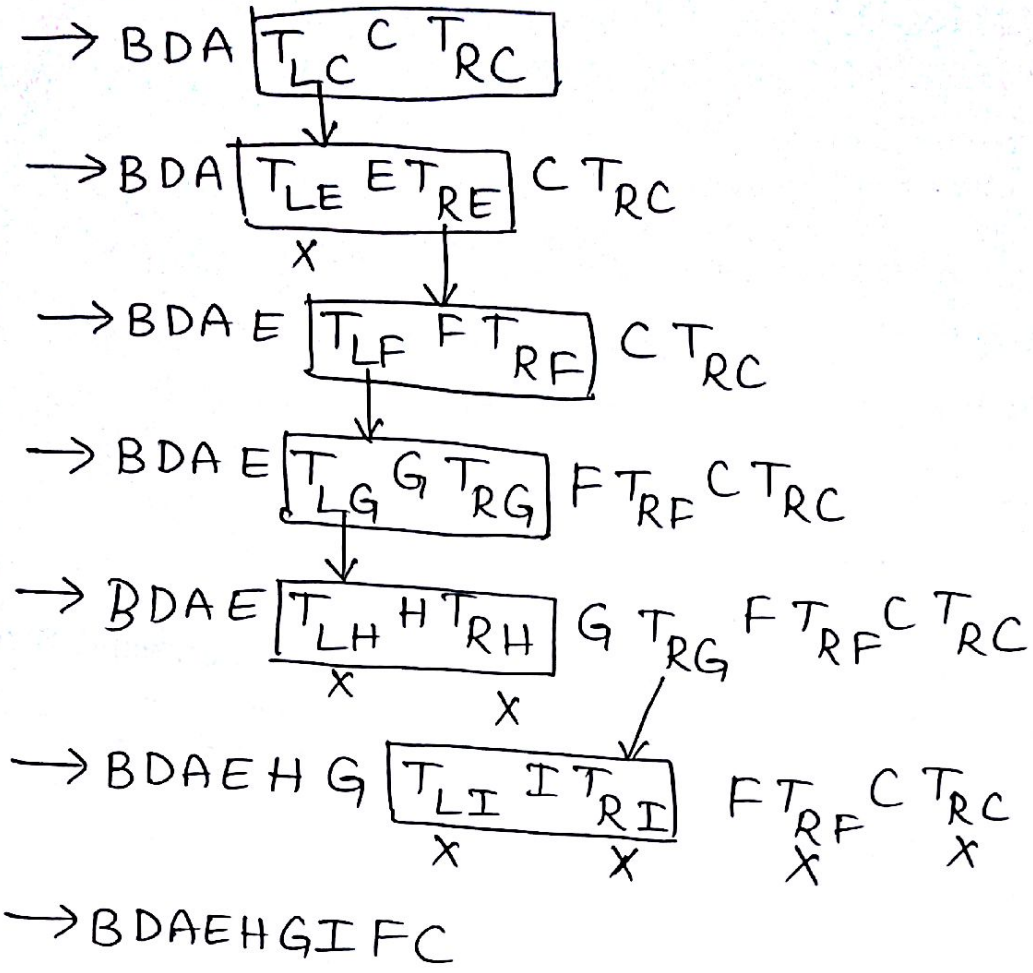



fig: Binary Tree

Inorder Traversal : LDR.








∴ The inorder for the given tree is BDAEHGIFC

(3) Postorder Traversal :-

→ Traverse the left sub tree of the root node R in postorder.

→ Traverse the right sub tree of the root node R in postorder.

→ visit the root node R.

Algorithm Postorder:

Input: ROOT is the pointer to the root node of the binary tree.

Output: Visiting all the nodes in postorder fashion

Data structure: Linked structure of binary tree

Steps:
1. ptr = ROOT
2. if (ptr ≠ NULL) then
3. postorder (ptr → LC)
4. postorder (ptr → RC)
5. Write ptr → DATA
6. end if
7. stop

ex:

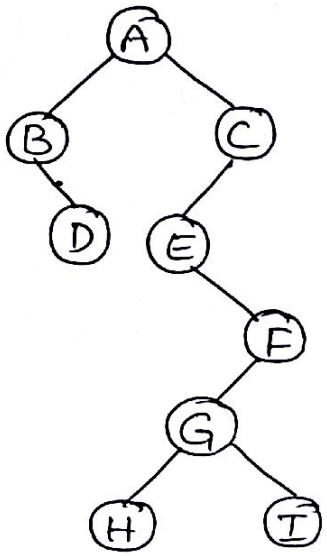



fig: Binary Tree



Postorder traversal : LRD

→ $T_{LA}\ T_{RA}\ A$

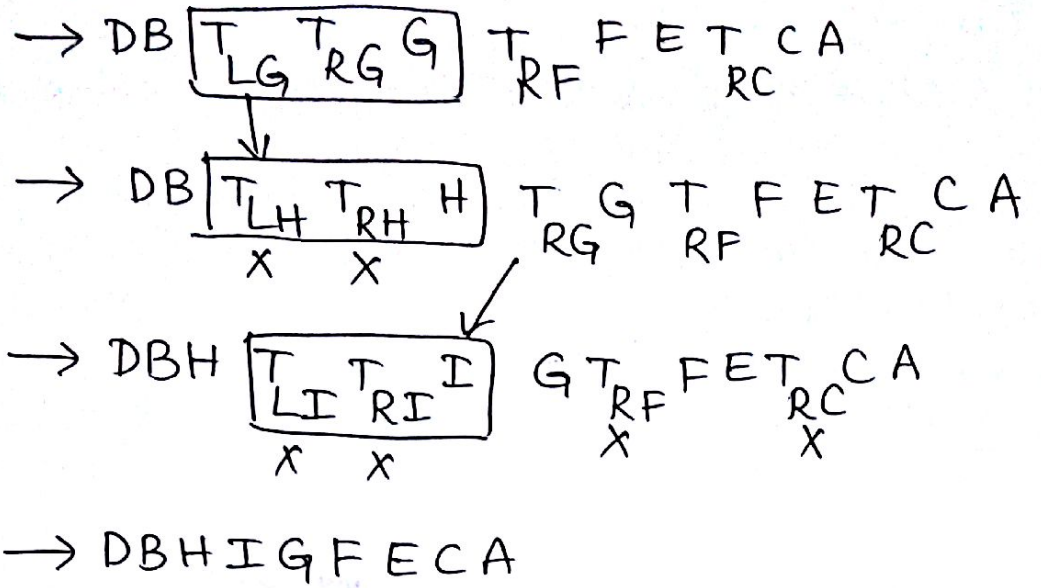
→ $T_{LB}\ T_{RB}\ B\ T_{RA}\ A$ (X)

→ $T_{LD}\ T_{RD}\ D\ B\ T_{RA}\ A$ (X X)

→ $D\ B\ T_{LC}\ T_{RC}\ C\ A$

→ $D\ B\ T_{LE}\ T_{RE}\ E\ T_{RC}\ C\ A$ (X)

→ $D\ B\ T_{LF}\ T_{RF}\ F\ E\ T_{RC}\ C\ A$




→ DBHIGFECA

∴ The postorder traversal for the given tree is,

DBHIGFECA.

Creation of Binary Tree from inorder & preorder Traversals :-